From States to Transitions: Improving Translation of LTL Formulae to Büchi Automata

Dimitra Giannakopoulou[1] and Flavio Lerda[2]

[1] RIACS/USRA, NASA Ames Research Center,
Moffett Field, CA 94035-1000, USA
dimitra@email.arc.nasa.gov
[2] School of Computer Science, Carnegie Mellon University,
Pittsburgh, PA 15213, USA
lerda@cmu.edu

Abstract. Model checking is an automated technique for checking that a system satisfies a set of required properties. With explicit-state model checkers, properties are typically defined in linear-time temporal logic (LTL), and are translated into Büchi automata in order to be checked. This paper describes how, by labeling automata transitions rather than states, we significantly reduce the size of automata generated by existing tableau-based translation algorithms. Our optimizations apply to the core of the translation process, where generalized Büchi automata are constructed. These automata are subsequently transformed in a single efficient step into Büchi automata as used by model checkers. The tool that implements the work described here is released as part of the Java PathFinder software (JPF), an explicit state model checker of Java programs under development at the NASA Ames Research Center.

1 Introduction

The use of LTL-based specifications in model checking is widespread. Many tools, including Java PathFinder, developed at the NASA Ames Research Center [1], and SPIN, from Bell Labs [2], after translating LTL formulae into Büchi automata, perform the verification using algorithms based on the one presented in [3]. These algorithms are linear in the size of the Büchi automata; however the Büchi automaton corresponding to an LTL formula may, in the worst case, be exponential in the size of the formula, making the model checking effort exponential in the size of the original formula. This worst-case complexity does not tend to occur for formulae of practical interest. Despite this fact, it is important for automata used for verification to be as small as possible, because memory is a major concern in model checking. Since finding the optimal sized Büchi automaton is a PSPACE-hard problem [4], the translation process becomes crucial in determining the size of the automata used for verification.

The translation of an LTL formula into a Büchi automaton that can be used for verification proceeds typically in three phases [4, 5]: 1) formula rewriting; 2) transla-

tion of the LTL formula into a generalized Büchi automaton – we refer to this as the "core" of the translation process; 3) conversion of the generalized Büchi automaton into a Büchi automaton, which can be used for model checking (we refer to this process as *degeneralization*). This paper focuses on phase 2, the core of the translation process. The existing algorithms proposed for this phase fall in two categories: those that are tableau-based ([6],[7]), and more recently an algorithm based on the construction of Alternating Automata [8], which claims better results for specific types of properties. The approach we take reduces the size of automata generated by the state-of-the-art in tableau-based algorithms [7]. Moreover, early comparisons of our translator against the one given in [8] on formulae that constitute the strength of their approach indicate that we perform equally well.

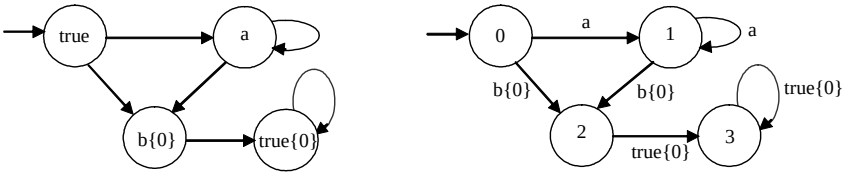

Fig. 1. Moving information from states to transitions. A label {0} denotes that a state or transition belongs to accepting set 0 of the automaton

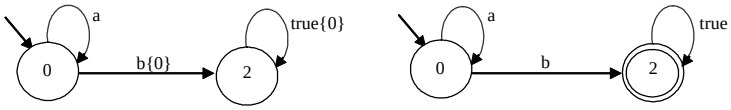

Fig. 2. TGBAs are translated to BA with accepting states (represented with a double circle)

Our improvement is based on the fact that our core algorithm generates transition-based generalized Büchi automata (TGBA), as opposed to state-based generalized Büchi automata (GBA). The algorithm of Gastin and Oddoux [8] also translates Alternating Automata into TGBAs. However, they attribute the improvements that their approach achieves to the use of Alternating Automata and to optimizations performed during intermediate stages of the translation. The use of TGBAs is not explicitly discussed as a factor in their results.

TGBAs carry labels on their transitions, whereas the GBAs generated by existing tableau-based algorithms carry labels on their states. Moreover, the accepting sets of TGBAs contain *transitions* rather than *states* (see Section 2 for exact definitions). Although TGBAs and GBAs are equally expressive, TGBAs allow us to obtain coarser partitions of equivalent states. For example, the first automaton in Fig. 1 is the automaton generated by [7] for formula a**U**b (**U** is the *strong until* operator). Note that states labeled with different propositions cannot be merged. By moving information to the transitions of the automaton, we obtain the second automaton of the figure. From this TGBA, one can see that states 0 and 2 are bisimilar with states 1 and 3, respectively. By merging these states, we obtain the first automaton in Fig. 2. This is the more compact automaton that our algorithm generates *directly*.

During the degeneralization phase, we efficiently translate *in a single step* the TGBA obtained from phase 2 into a non-generalized Büchi automaton with labeled transitions but accepting states, as used by model checking (see the second automaton of Fig. 2). This allows us to limit the growth of the number of states from the generalized to the non-generalized automaton.

The remainder of the paper is organized as follows. First, we provide background information in Section 2. A description of our algorithm for the core (phase 2) of the translation process is presented in Section 3, where the emphasis is on its differences from tableau-based algorithms. The degeneralization approach that we take is described in Section 4. Section 5 discusses the implementation of our tool, and Section 6 presents the results of our experiments comparing our approach to the state-of-the-art. Finally, Section 7 closes the paper with conclusions and future work.

2 Background

2.1 Linear Temporal Logic (LTL)

In this work, LTL is used to express temporal properties of a system for model checking. Given a set of atomic propositions $\wp$, a well-formed LTL formula is defined inductively using the standard Boolean operators, and the temporal operators **X** (next) and **U** (strong until) as follows:

- each member of $\wp$ is a formula,
- if φ and ψ are formulae, then so are $\neg\varphi$, $\varphi \vee \psi$, $\varphi \wedge \psi$, $\mathbf{X}\varphi$, $\varphi\ \mathbf{U}\ \psi$.

An interpretation for an LTL formula is an infinite word $w = x_0x_1x_2\ldots$ over $2^{\wp}$. In other words, an interpretation maps to each instant of time a set of propositions that hold at that instant. We write w_i for the suffix of w starting at x_i. LTL semantics is then defined inductively as follows ([6],[7]):

- $w \models p$ iff $p \in x_0$, for $p \in \wp$
- $w \models \varphi \vee \psi$ iff ($w \models \varphi$) or ($w \models \psi$)
- $w \models \varphi\ \mathbf{U}\ \psi$ iff $\exists\, i \geq 0$, such that: $w_i \models \psi$ and $\forall\, 0 \leq j < i$, $w_j \models \varphi$
- $w \models \neg\varphi$ iff not $w \models \varphi$
- $w \models \varphi \wedge \psi$ iff ($w \models \varphi$) and ($w \models \psi$)
- $w \models \mathbf{X}\,\varphi$ iff $w_1 \models \varphi$

We introduce the abbreviations "true $\equiv \varphi \vee \neg\varphi$" and "false $\equiv \neg$true". Temporal operators **F** (eventually) and **G** (always) typically used in LTL formulae are defined in terms of the main operators as follows: $\mathbf{F}\varphi \equiv$ true $\mathbf{U}\ \varphi$ and $\mathbf{G}\varphi \equiv \neg\mathbf{F}\neg\ \varphi$. As usual ([6],[7]), our algorithm works with formulae in *negation normal form*, that is, the $\neg$ operator is pushed inwards until it occurs only before propositions. To avoid an exponential blow up in the size of the translated formula, we define operator **V**, as follows: $\varphi\ \mathbf{V}\ \psi \equiv \neg(\neg\varphi\ \mathbf{U}\ \neg\psi)$. We refer to propositions and negated propositions as *literals*.

2.2 Büchi Automata

There are several variants of Büchi automata. The variant typically used in model checking is Büchi automata with labels on transitions and simple accepting conditions defined in terms of states (see Fig. 2, on the right). We will refer to these as *Büchi Automata*. However, for simplicity, translators first generate *generalized Büchi automata* (GBA), which have multiple accepting conditions. The core translation algorithms of most existing approaches ([6],[7]) produce *labeled* GBAs. These automata have labeled states and multiple accepting conditions defined in terms of states (see Fig. 1, left). Our approach and the approach presented in [8] produce *transition-based* GBAs. These have labeled transitions and multiple accepting conditions defined in terms of transitions (see Fig. 2, left). In what follows, we provide formal definitions for all three of these variants.

Definition 1 – A *Büchi automaton (BA)* is a 5-tuple $B = \langle S, A, \Delta, q_0, F \rangle$, where S is a finite set of states, A is a finite set of labels, $\Delta \subseteq S \times A \times S$ is a labeled transition relation, $q_0 \in S$ is the initial state, and $F \subseteq S$ is a set of accepting states.

An *execution* of the automaton B on an infinite word $w = a_0 a_1 a_2 \ldots$ over alphabet A is an infinite word $\sigma = s_0 s_1 s_2 \ldots$ over alphabet S, such that: $s_0 = q_0$ and $((s_i, a_i, s_{i+1}) \in \Delta, \forall i \in N)$. An infinite word w over alphabet A is *accepted* by the automaton B, if there exists an execution of B on w where some element of F occurs infinitely often.

Definition 2 – A *labeled generalized Büchi automaton (GBA)* is a 6-tuple $GB = \langle S, A, L, \Delta, q_0, F \rangle$, where S is a finite set of states, A is a finite set of labels, $L: S \rightarrow 2^A$ is a state labeling function, $\Delta \subseteq S \times S$ is a transition relation, $q_0 \in S$ is an initial state, and $F \subseteq 2^S$ is a set of sets of accepting states.

An *execution*

3 Translation Algorithm

In this section, we describe the core algorithm of the translation process. We postpone efficiency issues until Section 3.5, to facilitate a presentation of our algorithm (LTL2BUCHI) and that presented in [7] (LTL2AUT) within a common framework. As discussed in the introduction, the main feature of our algorithm is that it records information on transitions rather than states, which allows it to merge states that other approaches cannot. Note that we only include the basic temporal operators in our presentation; derived operators are transformed appropriately during parsing of the input formulae. For readers that are familiar with previous literature on the subject, we make our presentation follow the style of [6].

The translation algorithm is a tableau-like procedure, which builds a graph that defines the states and transitions of the automaton. Nodes in the graph are partitioned based on equivalence classes, where each equivalence class corresponds to a state. The nodes are labeled by sets of formulae, separated in formulae that need to be true immediately, and formulae that need to be true from the next state on. The algorithm processes formulae by expanding temporal operators based on the following fundamental identity: $\varphi \mathbf{U} \psi \equiv \psi \vee (\varphi \wedge \mathbf{X} (\varphi \mathbf{U} \psi))$. We start by introducing the data structure used by our algorithm.

3.1 Data Structure

The basic data structure that the automaton construction algorithm manipulates is the *Node*, which contains the following fields:

NodeId: A unique node id. Id 0 is reserved for the *initial* state.

Incoming: The incoming edges to the node, represented by the ids of the nodes with an outgoing edge leading to the current node.

ToBeDone: A set of formulae that must hold at the current node and have not been processed yet.

Old: A set of already processed *literals* that must hold at the current node.

Next: The set of formulae that must hold in all immediate successors of this node.

Eventualities: The set of promised and fulfilled eventuality obligations by the node. A promised obligation is a **U**-formula that has been processed in the current node, and a fulfilled obligation is a formula processed in the current node that is the right-hand side argument of some **U**-formula processed in the current node**.**

Accepting: The accepting sets to which the node belongs.

EquivClass: The id of the equivalence class to which the node belongs.

We keep a list of nodes, *nodes_set* whose construction has been completed, each having the same fields as described above. We denote the field *ToBeDone* of the node *q* by "*q.ToBeDone*", and similarly for other fields.

```
1  // expand is a method of class Node
2  ListofNodes expand (ListofNodes nodes_set) {
3  if This.ToBeDone is empty { // node has been fully processed
4      compute_accepting(This);
5      if ∃ ND ∈ nodes_set s.t. equivalent(This, ND) {
6          // this node is equivalent to a node that has already been computed
7          merge(ND, This, nodes_set);
8          return nodes_set;
9      } // end if
10     else { // processed node to be added to nodes_set
11         This.EquivClass = new_class_id();
12         nodes_set = nodes_set ∪ {This};
13         create NewNode with: {NodeId = new_node_id(),
14            Old=Next=Accepting = {}, Incoming = This.NodeId,
15            ToBeDone = This.Next};
16         return NewNode.expand(nodes_set);
17     }
18 } else { // ToBeDone is not empty, so keep processing
19      let next_formula ∈ This.ToBeDone;
20      This.ToBeDone = This.ToBeDone \ {next_formula};
21      update_fulfilled_obligations(This, next_formula);
22
23      if (contradicts(next_formula, This))
24          return nodes_set;    // node gets discarded
25      if (isRedundant(next_formula, This)) // formula is redundant
26          return This.expand(nodes_set);     // no need to process it
27      if (next_formula is a U formula)
28         update_promised_obligations(This, next_formula);
29
30      // no contradictions, and formula is not redundant, so we process it
31      if  (next_formula is not a literal) {
32          if (next_formula is a 'U', 'V' or '∨' formula) {
33               Node2 = This.Split(next_Formula); // split in 2 nodes
34               return Node2.expand(This.expand(nodes_set));
35          }
36          if (next_formula is a 'φ ∧ ψ' formula) {
37               ToBeDone = ToBeDone ∪ ({φ,ψ} \ Old);
38               return This.expand(nodes_set);
39          }
40          if (next_formula is a 'X φ' formula) {
41               Next = Next ∪ (φ};
42               return This.expand(nodes_set);
43          }
44      } else { // next formula is a literal
45               Old = Old ∪ {next_formula};
46               return This.expand(nodes_set);
47       }
48   } // end of "else ToBeDone not empty"
49 }
```

Fig. 3. Node expansion algorithm

3.2 Generic Algorithm

The core of algorithms LTL2BUCHI and LTL2AUT is method *expand*, which is illustrated in Fig. 3. The two algorithms differ in the way they implement the following methods: *compute_accepting*, *equivalent*, *merge*, *update_fulfilled_obligations*, and *update_promised_obligations*. The algorithm for translating a formula φ starts by creating an initial node *INIT* with *NodeId* = *EquivClass* = 0, with *Next* = {φ}, and with all other fields empty. The list of nodes *nodes_set* is initially empty. The translation of φ is performed by calling *INIT.expand(nodes_set)* to expand node *INIT*. Let us at this stage describe how the expansion method works. The line numbers in the following description refer to the algorithm that appears in Fig. 3. With the current node, the algorithm first checks if there are unprocessed obligations left in *ToBeDone*. We examine two cases: 1) there are *none* 2) there are some obligations left in *ToBeDone*.

Case 1 – there are no obligations left in *ToBeDone* (lines 3-17). The fact that *ToBeDone* is empty shows that the current node is fully processed, and ready to be added to *nodes_set*. It also contains enough information for us to compute the accepting sets to which it belongs (line 4). If this node is equivalent to an existing node in *nodes_set*, then the two nodes are merged (line 7), meaning that the information they hold is combined appropriately. If it is not equivalent to any existing node (lines 10-17), it defines a new state of the automaton generated. A new equivalence class Id is assigned to the node's *EquivClass* field, and the node is added to *nodes_set*. Moreover, a new node *NewNode* is created as the immediate successor of the current node (the one that has just been processed). The *Incoming* field of *NewNode* is the *NodeId* of the current one, and its obligations in *ToBeDone* are the obligations that the current node holds in its *Next* field. In other words, after a node is processed, its future obligations are delegated to its immediate successors.

Case 2 – there are obligations left in *ToBeDone* (lines 18-48). If the current node contains obligations in *ToBeDone*, a formula *next_formula* is removed from this set. Method *update_fulfilled_obligations* is called, to record potential eventualities that the formula that is being processed fulfills. We then check whether this formula contradicts any information contained in the node. If a contradiction occurs, it means that the node must be discarded. If no contradiction occurs, then we check whether the formula is redundant (i.e. covered by some information contained in this node), in which case we simply do not need to process it. If the formula is not redundant, and is a **U**-formula, it promises to fulfill some obligation in the future (since **U** is a strong until), which is recorded by method *update_promised_obligations*.

If the formula is not redundant and it does not contradict existing node information, then it gets processed as follows. When *next_formula* is a literal, then the formula is simply added to field *Old* of the node (lines 44-47). When *next_formula* is not a literal (lines 31-43), the current node is either split in 2 nodes or not split, and new formulae may be added to the fields *ToBeDone* and *Next*. (Note that when we split a node, for efficiency reasons, we do not create two new nodes, but modify the current one, and

create an additional node.) The exact actions performed depend on the form of *next_formula* and are the following:

- *next_formula* = $\varphi \wedge \psi$. Then φ and ψ are added to *ToBeDone* because they both need to be true for the formula to hold.
- *next_formula* = $\mathbf{X}\varphi$. Then φ is added to field *Next*.
- *next_formula* is in either of the forms: $\varphi \vee \psi$, $\varphi \mathbf{U} \psi$, $\varphi \mathbf{V} \psi$. There are two alternative ways of making these formulae true. So the node is split into two nodes, each representing one way of making the formula true. For $\varphi \vee \psi$, φ is added to *ToBeDone* of one node, and ψ to that of the other. For $\varphi \mathbf{U} \psi$, φ is added to *ToBeDone* and $\varphi \mathbf{U} \psi$ to *Next* of one node, and ψ is added to *ToBeDone* of the other. This splitting can be explained by observing that $\varphi \mathbf{U} \psi$ is equivalent to $\psi \vee (\varphi \wedge \mathbf{X} (\varphi \mathbf{U} \psi))$. For $\varphi \mathbf{V} \psi$, ψ is added to *ToBeDone* of both nodes, φ is added to *ToBeDone* of one node, and $\varphi \mathbf{V} \psi$ to *Next* of the other. This splitting can be explained by observing that $\varphi \mathbf{V} \psi$ is equivalent to $\psi \wedge (\varphi \vee \mathbf{X} (\varphi \mathbf{V} \psi))$.

The splitting algorithm is illustrated in Fig. 4. Table 1 illustrates, for the types of formulae that cause a node to split, the formulae added to fields of the resulting nodes (although $\wedge$ formulae are not split, we include an entry in the table, because the fields in this table are also used for the definition of syntactic implication used in the computation of redundancies and contradictions). For example, the formulae in *New1* and *New2* are added to the *ToBeDone* field of the first and second resulting node, respectively. Moreover, *Next1* is added to the *Next* field of the first resulting node.

Table 1. Definition of New and Next functions for non-literals

form	New1(form)	Next1(form)	New2(form)
$\varphi \mathbf{U} \psi$	$\{\varphi\}$	$\{\varphi \mathbf{U} \psi\}$	$\{\psi\}$
$\varphi \mathbf{V} \psi$	$\{\psi\}$	$\{\varphi \mathbf{V} \psi\}$	$\{\varphi, \psi\}$
$\varphi \vee \psi$	$\{\varphi\}$	$\varnothing$	$\{\psi\}$
$\varphi \wedge \psi$	$\varnothing$	$\varnothing$	$\{\varphi, \psi\}$

//split is a method of class Node. It splits a node into two, using information in Table 1

```
Node split (Formula form) {
     create Node2 with new Id but otherwise identical to This;
     Node2.ToBeDone = This.ToBeDone ∪ (New2(form) \ Old);
     // modify This (current node) as follows
     This.ToBeDone = This.ToBeDone ∪ (New1(form) \ Old);
     This.Next = Next ∪ Next1(form);
     return Node2;
}
```

Fig. 4. The splitting algorithm

The copies are processed in DFS order, i.e., when expansion of the current node and its successors are finished, the expansion of the second copy and its successors is

started. Note that a formula is only added to *ToBeDone* if it does not exist in *Old* – hence the fact that we take (*New(form)* \ *Old*) (see Fig. 4). This is purely for efficiency, that is, to avoid processing a formula that has already been processed. Also, the fields *Incoming, ToBeDone, Old,* and *Next* of each node are sets, and therefore contain no duplicates.

Testing for contradictions and redundancies. As described by [7], the checks for contradiction and redundancy are based on deriving the set of formulae SI*(A, B)* that are syntactically implied from sets of formulae *A* and *B*, where *B* represents formulae that have to hold at the next state. We use the following inductive definition:

1. TRUE ∈ SI*(A, B)*,
2. μ ∈ SI*(A, B)*, if μ ∈ *A*,
3. μ ∈ SI*(A, B)*, if μ is not a literal and either of the following hold:
   - (New1(μ) ⊆ SI*(A, B)*) and (Next1(μ) ⊆ *B*)
   - New2(μ) ⊆ SI *(A, B)* (see Table 1 for definitions of New and Next)

A formula φ contradicts a node *nd*, if ¬φ ∈ SI*(nd.Old, nd.Next)*. In other words, *contradicts(φ, nd)* returns true if ¬φ ∈ SI*(nd.Old, nd.Next)*. A formula φ is redundant for a node *nd*, if φ ∈ SI*(nd.Old, nd.Next)*, and additionally, if φ is μ **U** ν (that is, φ is a **U**-formula), then ν ∈ SI*(nd.Old, nd.Next)*. As mentioned in [7], the special attention to the right-hand side arguments of **U**-formulae is to avoid discarding information required to define accepting conditions. So to summarize, *isRedundant(φ, nd)* returns true if (φ ∈ SI*(nd.Old, nd.Next)*) and (either φ is not a **U**-formula, or φ's right hand argument ν ∈ SI*(nd.Old, nd.Next)*).

3.3 Obtaining the Automaton

The *nodes_set* returned by the expand algorithm is used to construct a TGBA that corresponds to the formula translated. There are as many states in the automaton as there are different *EquivClass* ids in the corresponding field of nodes in *nodes_set*. A state with id *i* represents all nodes *nd* in *nodes_set* such that *nd.EquivClass =i.* A transition *trans* exists between state *i* and state *j* iff: {∃ nodes nd_i, nd_j such that nd_i*.EquivClass = i* and nd_j*.EquivClass = j* , and nd_i ∈ nd_j*.Incoming*}. The propositions that label *trans* are nd_j*.Old*, which means that the transition is triggered iff all literals in nd_j*.Old* hold. Finally, the accepting sets to which *trans* belongs are defined by nd_j*.Accepting.* We thus obtain a transition-based generalized Büchi automaton, which is subsequently degeneralized as described in Section 4.

3.4 Tuning for LTL2AUT or LTL2BUCHI

Our description of the algorithm so far applies both to LTL2AUT and LTL2BUCHI. In this section, we show how implementations of the remaining functions can be tuned to obtain each of the two algorithms.

LTL2AUT. Two nodes are equivalent if their *Old* and *Next* fields are equal, i.e., *equivalent(Node nd1, Node nd2)* returns *true* if (*nd1.Old* equals *nd2.Old*) and (*nd1.Next* equals *nd2.Next*). Merging two nodes updates the *Incoming* field of the first node to the union of their respective *Incoming* fields, i.e. *merge(Node nd1, Node nd2, ListOfNodes nodes_set)* performs { $nd1.Incoming = nd1.Incoming \cup nd2.Incoming$}.

LTL2AUT does not use the *Eventualities* field. Rather, it computes the accepting conditions that a node fulfills in terms of fields *Old* and *Next*. As a result, methods *update_fulfilled_obligations* and *update_promised_obligations* have empty bodies. Method *compute_accepting(Node nd)* adds the accepting set defined by a formula $\varphi\mathbf{U}\psi$ to *nd.Accepting* if the following condition holds: ($\varphi\mathbf{U}\psi \in SI(nd.Old, nd.Next)$) $\rightarrow$ ($\psi \in SI(nd.Old, nd.Next)$).

LTL2BUCHI. Two nodes are equivalent if their *Next* fields are equal, i.e., *equivalent(Node nd1, Node nd2)* returns *true* if (*nd1.Next* equals *nd2.Next*). Merging two nodes first checks if fields *Old* and *Accepting* of the nodes are equal. If they are, then the *Incoming* field of the first node gets updated to the union of their respective *Incoming* fields. Otherwise, the *EquivClass* field of the second node is updated to that of the first node, and the second node is added to the *nodes_set*:

```
merge(Node nd1, Node nd2, ListOfNodes nodes_set) {
     if ((nd1.Old equals nd2.Old) and
                (nd1.Accepting equals nd2.Accepting))
     then {nd1.Incoming = nd1.Incoming ∪ nd2.Incoming);
     else {
         nd2.EquivClass = nd1.EquivClass;
         nodes_set = nodes_set ∪ {nd2};
     }
}
```

So merging in the context of LTL2BUCHI consists of recording the fact that two nodes are equivalent, and adding the information that the new node provides to the *nodes_set*. The reason why we cannot always merge nodes in the same way as LTL2AUT is that the *Old* and *Accepting* fields of the two nodes may differ, meaning that the state in the final automaton that corresponds to this equivalence class can be obtained by different transitions. By recording the information of both nodes, we are able, at the end of the construction, to label these transitions appropriately.

As soon as a formula is processed in a node, if this formula is the right-hand side argument of a **U**-formula (**U**-formulae define accepting sets in the automaton generated), then this formula must be added to field *Eventualities* of the node. So method *update_fulfilled_obligations(Node nd, Formula form)* performs { if (*form* is right-hand side argument of a ***U-formula***) $nd.Eventualities = nd.Eventualities \cup \{form\}$ }. If a **U**-formula is processed, unless it is redundant, the node promises to fulfill its obligations, so *update_promised_obligations(Node nd, Formula form)* performs {$nd.Eventualities = nd.Eventualities \cup \{form\}$}.

Finally, *compute_accepting(Node nd)* computes *nd.Accepting* as follows: the accepting set corresponding to a formula $\varphi \mathbf{U} \psi$ belongs to *nd.Accepting* iff $\{(\varphi \mathbf{U} \psi \in$ *nd.Eventualities*$) \rightarrow (\psi \in$ *nd.Eventualities*$)\}$.

3.5 Optimizations

The above presentation of our algorithm focused on simplifying its comparison with that presented in [7]. In this section, we discuss how LTL2BUCHI has been implemented for efficiency. Implementation details can be found in [10].

Let φ be the formula to be translated. Each accepting set for φ is assigned a unique integer in the range 0..(#accepting_sets – 1). The set of accepting sets to which a node *nd* belongs is then represented as a bitmap *B* of size #{accepting_sets}. If location *i* of *B* is set, then *nd* belongs to accepting set *i*. An abstract syntax graph is initially generated for formula φ. This graph is similar to an abstract syntax tree, but is a directed acyclic graph (dag) where equal formulae appear only once, i.e., they are represented by a single node in the graph. That allows for fast comparison of formulae in the subsequent phases of the algorithm, based on equality of their references. Each node of the graph represents a sub-formula of φ (with the root referencing φ itself).

For each (sub)-formula *form*, the following information is recorded. If *form* is a **U**-formula, we record the index (*untils_index*) of the accepting set that *form* defines. This index is obtained by calling *form.get_untils_index()*. If *form* is the right-hand side argument of one or more **U**-formulae μ_i, we record for *form* in a bitmap *RightofWhichUntils* the indices of all μ_i formulae. In other words, if bit *i* is set in the *RightofWhichUntils* field of formula *form*, it means that *form* is the right-hand side argument of the **U**-formula that defines the i^{th} accepting set. This bitmap is obtained by calling *form.get_rightOfWhichUntils()*.

Field *Eventualities* of a node *nd* does not record formulae explicitly. Rather, it is made up of two bitmaps, one for the *promised* obligations, and one for the *fulfilled* obligations. Whenever a **U**-formula is processed in a node, the bit whose index corresponds to this formula must be set in *Eventualities.promised*. Whenever a formula that is the right-hand side argument of some **U**-formulae is processed, then the bits corresponding to these **U**-formulae must be set in *Eventualities.fulfilled*. Updating this information on the node is therefore performed as follows:

```
update_fulfilled_obligations(Node nd, Formula  form) {
      nd.Eventualities.fulfilled = (nd.Eventualities.fulfilled
            bitwise_or form.get_rightOfWhichUntils()); }

update_promised_obligations(Node nd, Formula  form) {
    set bit form.get_untils_index() in
            nd.Eventualities.promised; }
```

Finally, the *Accepting* field of a node is computed as follows:

```
compute_accepting(Node nd) {
nd.Accepting = (bitwise_not (nd.Eventualities.promised))
            bitwise_or nd.Eventualities.fulfilled; }
```

Therefore, we avoid computing the accepting conditions of a node in terms of syntactic implications, as is done by [7], since this information is obtained during the actual expansion. We also avoid storing explicitly all relevant formulae that are processed, as performed in [6], but rather store them in terms of bitmaps, which also allow us a very efficient way of obtaining the accepting conditions in the end.

In addition to the above, during the translation of a formula, we represent a partition of equivalent nodes in a more compact form, as described in [10]. Fully processed nodes are stored in a data structure, which we call a *state*. Each state represents an actual state of the final automaton, i.e. an equivalence class of nodes. In field *Transitions*, a state stores information about the set of transitions that lead to that state, and in *Next*, it stores the formulae that need to hold for its immediate successors. Each transition holds information about its source states, its labels, and the accepting sets to which it belongs.

3.6 Correctness Argument

We argue for the correctness of our approach by comparing it to other approaches proven correct. In essence, the core of our algorithm consists of applying the algorithm of [7] to generate transition-based generalized Büchi automata. There is therefore a direct correspondence between the automata we generate and those generated by the [7] algorithm. One could move between equivalent automata of each type by just applying simple transformations, if one assumes that the information about the *Next* fields of nodes is maintained in states of these automata. All that our algorithm does is that, by moving labels to transitions, it can always merge states that have the same future (same *Next* fields).

Our computation of the accepting sets to which states/transitions belong is based on [6]. The idea there is to determine, for each **U** formula that has been processed in a *Node*, whether its right-hand side argument has also been processed. In other words, whether the eventualities promised are fulfilled. However, similarly to [7], we do not process some redundant formulae. We therefore need to decide how to treat redundant formulae that are **U**-formulae or right-hand side arguments of **U**-formulae. From the definitions of Section 3.2, for a **U**-formula to be redundant, its right-hand side argument must be syntactically implied by the current *Node*. According to [7], this *Node* is then accepting with respect to the set that the **U**-formula defines. For this reason, when a **U**-formula is redundant, we do not set its bit in the *promised* bitmap. On the other hand, it is possible for the right-hand side of a **U**-formula to be redundant, but for its corresponding **U**-formula to have been processed in the *Node* that is being expanded. To cover for this possibility, we always update the *fulfilled* bitmap when the right-hand side of a **U**-formula is encountered. Our scheme for computing accepting transitions thus avoids re-computing syntactic implications in order to establish accepting states/transitions as is performed in [7], but also avoids storing explicitly all relevant formulae that are processed, as performed in [6].

4 Degeneralization

The TGBAs generated as described in Section 3 must be translated to a Büchi automaton. We perform this by using Büchi automata that we call *degeneralizers*. Degeneralization algorithms are well known in the literature but for completeness, we include here the way (inspired by [8]) in which we generate and use degeneralizers in our tool.

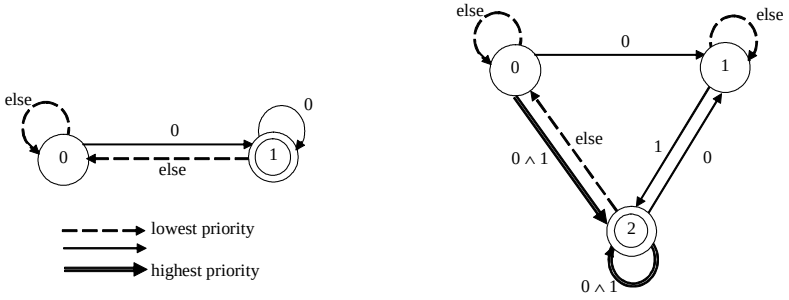

Fig. 5. On the left/right, we illustrate the degeneralizer used for a generalized automaton with one/two accepting sets, respectively

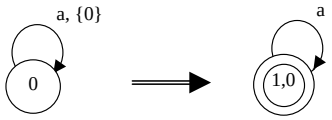

Fig. 6. Degeneralization of the TGBA for "**G**a", using the automaton on the left of Fig. 5

A degeneralizer for a TGBA is a *deterministic* Büchi automaton. Degeneralizers have a fixed number of states for a fixed number of accepting sets in their targeted TGBAs. They express the fact that a path in the TGBA is accepting if it contains infinitely often at least one accepting transition from each one of the accepting sets. The transitions of a degeneralizer DG are therefore labeled with predicates that relate to accepting sets. A BA is obtained by a TGBA by computing its synchronous product with the appropriate degeneralizer. That is, a joint transition (t_1, t_2) of a DG with a TGBA is enabled if t_2 belongs to the accepting set that the predicate on t_1 requires. The accepting states of the product are the ones where the degeneralizer is in an accepting state. Fig. 6 illustrates the result of degeneralizing the GBA generated for formula **G***a* with the degeneralizer depicted on the left of Fig. 5, where state "1" is the initial state.

Fig. 5 illustrates degeneralizers for automata with one and two accepting sets. These degeneralizers are *deterministic*; transitions are explored based on their priority – lower priority transitions are only explored if higher priority ones cannot fire. *Else* transitions are taken when no other outgoing transitions from a state are eligible (they have the lowest priority). A degeneralizer for *n* accepting sets is generated with the algorithm of Fig. 7. This algorithm generates labels in the order in which they should

be explored, i.e. it generates higher priority transitions first. Note that transitions that have more requirements have higher priority, with "else" transitions having the lowest.

From Fig. 5, it is clear that degeneralizers only accept infinite words that satisfy infinitely often the predicates related to each accepting set. However, such automata impose an order on the way these accepting sets should be satisfied. This order allows us to have deterministic degeneralizers, but the size of the Büchi automata generated is sensitive to this order. We are interested in coming up with heuristics that determine what degeneralizer would be better for a TGBA. Currently, our tool performs the degeneralization using a degeneralizer with the initial state being state 0 or the accepting state, and returns the smaller of the two results. Although very simple, this approach allows us to avoid unnecessary growth of the TGBA during degeneralization, especially for TGBAs with a small number of accepting sets.

```
Büchi generate (int acc_sets) {
    nnodes = acc_sets + 1; // number of automaton nodes
    last = acc_sets; // last automaton node
    for (int i=0; i < nnodes; i++)
        create automaton state Si;
    for (int i=0; i <last; i++) {
        for (int j=last; j>i; j--) {
            create transition "trans" from Si to Sj;
            for (int k=i; k<j; k++)
                 add label k to trans;
         }
        create looping transition for Si labeled with else;
    }
    // now dealing with last node
    create looping transition trans for Slast;
    for (int i=0; i<last; i++)
          add label i to trans;

    for (int i=last-1; i>=0; i--) {
        if (i == 0) then
            create transition from Slast to Si labeled else;
        else {
            create transition trans from Slast to Si;
            for (int j=0; j<i; j++)
                add label j to trans;
        }
    }
}
```

Fig. 7. Algorithm that generates degeneralizers

5 Implementation

The algorithms presented here have been implemented in Java in the LTL2BUCHI tool, which is being distributed with the JavaPathfinder (JPF) software from NASA Ames

Research Center (please contact the first author for details). LTL2BUCHI has been written in Java for readability, portability, but also to be integrated with JPF. The work that was presented here is however completely general, since the translation does not need to take into account the syntax of the atomic predicates, which is the only part specific to JPF. LTL2BUCHI is efficient, and generates very concise automata for most LTL properties of practical interest.

Formulae that apply to Java software tend to be more complicated than for typical modeling languages, since they may involve such issues as the creation of objects. So similarly to [8], our tool performs optimizations after each translation phase, to keep the automata generated small in size. Moreover, we try to keep our translator up-to-date with respect to new optimizations in the literature. Currently, LTL2BUCHI covers a large part (but not all) of existing optimizations. A brief list of the ones currently implemented is the following (these are described in detail in [10], where we also explain how optimizations defined for GBA can be applied to TGBA):

1. Optimizations to initial formula by rewriting. We have implemented a simple rewriting engine that includes several rewriting rules presented in [4, 5].
2. Optimizations to the TGBA before degeneralization. We call "SCC-optimizations" a family of optimizations that are based on computing the strongly-connected components of the automaton graph. These are based on optimizations defined by [4, 5], but are applied to transition-based generalized Büchi automata. A very simple, but very useful optimization that we also perform is to ignore sets of accepting sets that are supersets of other accepting sets. We call this the "superset reduction" [10].
3. Optimizations to the BA. These include SCC-optimizations [4, 5], and bisimulation reduction followed by fair simulation reduction [4].

6 Experiments

We have performed extensive tests between our algorithm, LTL2BUCHI, and the state-of-the-art in tableau-based translation algorithms, LTL2AUT [7]. Note that, with the exception of the work by Oddoux and Gastin [8] that also affects the "core" of the translation process, all other optimizations defined in the literature ([4], [5]) apply to the other phases of this process, and are thus complementary to ours. The main characteristic of our approach is that it cannot generate larger automata than LTL2AUT, since, in the worst case, each node in the graph generated defines a separate equivalence class. Moreover, since our algorithm produces coarser equivalence classes of states in general, it is also expected to generate fewer transitions. So our experiments have focused on quantifying the reduction that LTL2BUCHI achieves in practice.

To compare LTL2BUCHI with LTL2AUT we have used an implementation of LTL2AUT in our tool. This is for two reasons. First, so as to record the performance of the two algorithms in the same framework, and during all phases of the translation (i.e., core translation, degeneralization, optimizations discussed in Section 5). Second,

because our tool implements some small optimizations not presented here; since our comparisons focus on the fact that we use TGBA, the rest of the optimizations have to be exactly the same.

6.1 Experimental Set-Up

The comparison has been performed in terms of a set of tests on random formulae, inspired by [7]. More specifically, a set of tests contains *F* randomly generated formulae of length *L* with *N* propositional variables. Parameter *P* describes the probability of generating the temporal operators **U** and **V** in the formulae. Each algorithm is run on a set of tests, and the average performance over the *F* formulae is computed, in terms of number of states and transitions of the automata generated.

To generate a formula of size *L*, we use the following algorithm. For *L=1*, we generate a random propositional variable using a uniform distribution. For *L=2*, we generate a random unary operator from the set {$\neg$, **X**}, using a uniform distribution, and we apply it to a random formula of size 1. For *L>2*, we generate a random operator from the set {$\neg$, **X**, $\vee$, $\wedge$, **U**, **V**}, with a probability equal to *P/2* to generate either **U** and **V**, and *(1-P)/4* for the other operators. If the chosen operator is unary, it is applied to a random formula of size *L-1*; if it is binary, it is applied to a random formula of size *S* and one of size *L-S-1*, where *S* is randomly chosen between *1* and *L-2* inclusive, and with a uniform distribution.

6.2 Results

We have performed tests for values of *N* equal to *3* and *5*. In the first set of tests, the probability *P* is set to *1/3*, that is, all operators have an equal probability of being generated. We then increase the probability of generating temporal operators, by setting *P* to *1/2*. We plot the results as a function of the length *L* of the formulae, where *L* ranges from *5* to *30*, in increments of *5*. We compare the size of both the generalized, and the degeneralized automata generated by the two algorithms. Finally, we also perform a test with all the optimizations described in Section 5 turned on.

In all cases our algorithm outperforms LTL2AUT with respect to both the number of states and the number of transitions, for the generalized as well as for the final Büchi automata. It can also be noted that the improvement in number of states is approximately the same as that obtained for the number of transitions. Except for the case where optimizations are used, the improvement on the generalized Büchi automata is conserved almost unaltered after degeneralization.

The first row of Fig. 8 shows the ratio between the results obtained with our algorithm and the LTL2AUT algorithm, without applying any optimizations (we compare the "core" of the translation): for each data point it depicts the average number of states and transitions, before and after degeneralization. As the length of the formulae increases the improvement obtained over the LTL2AUT algorithm becomes more significant, and so does it for a larger number *N* of propositional variables. This can be

attributed to the fact that the complexity of a formula increases with its length and the number of propositional variables it contains, leaving more room for improvement in the translation. On the left side of the second row, we analyzed the behavior for random formulae generated assigning a higher probability to temporal operators. As it can be seen, the improvement again becomes more pronounced, showing that the algorithm still performs better in the presence of more temporal operators.

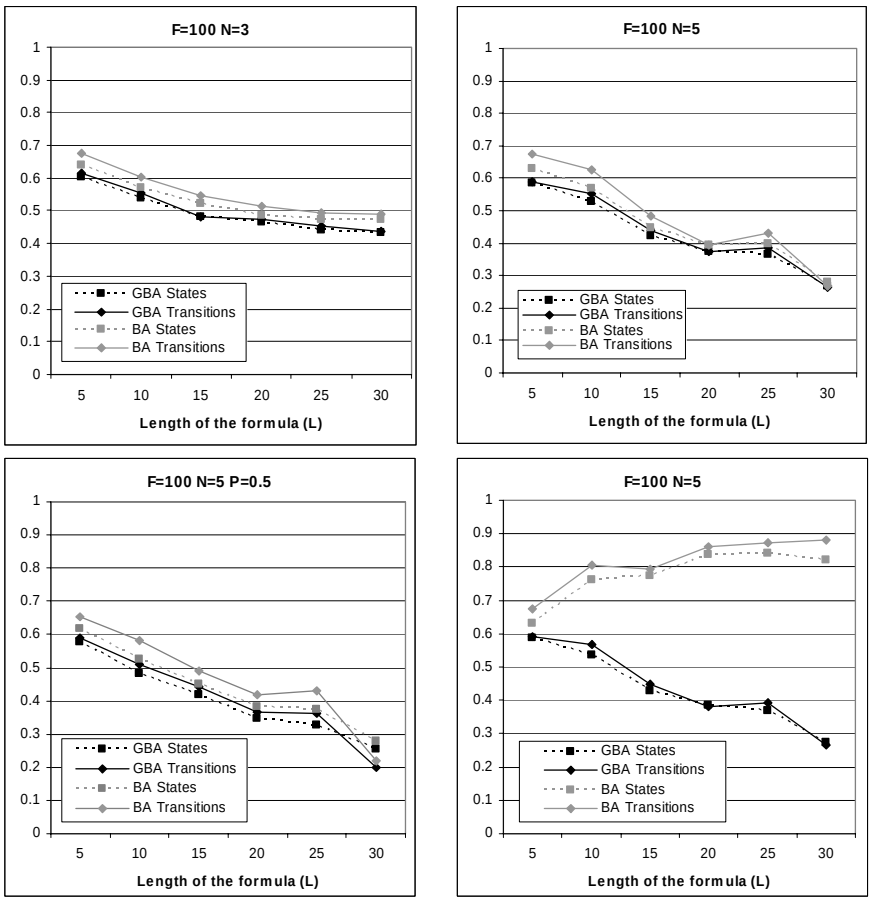

Fig. 8. Ratio of states and transitions generated by the LTL2BUCHI algorithm over the ones generated by the LTL2AUT algorithm, for the generalized and final Büchi automata. The first row displays results for *F = 100*, *P=1/3*, *N=3* (left) and *N=5* (right). The second row displays results for *F=100* and *N=5*, but for increased probability *P=1/2* of generating temporal operators (left), and for the case where all optimizations are turned on (right).

The right side of the second row illustrates the results obtained when all optimizations are enabled: these include rewriting before the formula is translated and optimi-

zations of the generated automaton. By comparing the results to those on the right of the first row in terms of the generalized Büchi automata, it is clear that rewriting has the same effect on both algorithms. On the other hand, optimizations of the generated automata reduce the overall improvement significantly. This is as expected, because our tools implement quite strong optimizations (including bisimulation, and fair simulation [4]). However even such strong optimizations do not completely compensate for the improvements that LTL2BUCHI achieves over LTL2AUT. Moreover, it is obvious that reducing the size of the automata early on makes the application of the optimization algorithms less expensive, both in terms of space, and time.

7 Conclusions

We have presented an algorithm that significantly improves the state-of-the-art in tableau-based translations of LTL formulae into Büchi automata, by computing transition-based, rather than state-based automata. The possibility of moving information to transitions has also been pointed out by Wolper in his tutorial paper [11], although no algorithm is provided. Transition-based automata are also used in [8], although in the context of using Alternating Automata in the translation process. An advantage of our optimization is that it is based on very simple modifications to the "core" of the tableau-based translation process, and could therefore be easily implemented by existing tableau-based translators. We have demonstrated the improvements achieved by our algorithm through extensive testing.

In the future, we intend to perform more systematic testing and comparison of our tool with existing ones. We have already performed preliminary experiments with formulae that are typically used in verification and with formulae that are used to stress test such translators. For example, in testing it against "fairness formulae", on which [8] report that their translation performs better than existing approaches, our tool generates automata with the same number of states (with optimizations turned on for both tools). As compared to SPIN [2] and to the tool by Etessami [4], our translator generates more compact automata.

Finally, we are investigating ways of further improving our translation algorithms. For example, we wish to perform more efficient comparisons between the *Next* fields of the nodes during expansion, to allow further reductions. We also intend to update our tools with other optimizations defined in the literature.

Acknowledgements

We wish to thank Howard Barringer for his feedback on earlier versions of this paper and Cindy Kong for implementing the fair simulation algorithm.

References

[1] Visser, W., Havelund, K., Brat, G., and Park, S. "Model Checking Programs", in *Proc. of the 15th IEEE International Conference on Automated Software Engineering (ASE'2000)*. 11-15 September 2000, Grenoble, France. IEEE Computer Society, pp. 3-11. Y. Ledru, P. Alexander, and P. Flener, Eds.

[2] Holzmann, G.J., *The Model Checker SPIN*. IEEE Transactions on Software Engineering, Vol. **23**(5), May 1997: pp. 279-295.

[3] Courcoubetis, C., Vardi, M., Wolper, P., and Yannakakis, M., *Memory-Efficient Algorithms for the Verification of Temporal Properties.* Formal Methods in System Design, Vol. **1**, 1992: pp. 275-288.

[4] Etessami, K. and Holzmann, G. "Optimizing Buechi automata", in *Proc. of the 11th International Conference on Concurrency Theory (CONCUR'2000)*. August 2000, Pennsylvania, USA. Springer, LNCS 1877.

[5] Somenzi, F. and Bloem, R. "Efficient Buechi automata from LTL Formulae", in *Proc. of the 12th International Conference on Computer Aided Verification (CAV 2000)*. July 2000, Chicago, USA. Springer, LNCS 1855. E.A. Emerson and A.P. Sistla, Eds.

[6] Gerth, R., Peled, D., Vardi, M.Y., and Wolper, P. "Simple On-the-fly Automatic Verification of Linear Temporal Logic", in *Proc. of the 15th IFIP/WG6.1 Symposium on Protocol Specification, Testing and Verification (PSTV'95)*. June 1995, Warsaw, Poland.

[7] Daniele, M., Giunchiglia, F., and Vardi, M.Y. "Improved Automata Generation for Linear Temporal Logic", in *Proc. of the 11th International Conference on Computer Aided Verification (CAV 1999)*. July 1999, Trento, Italy. Springer, LNCS 1633.

[8] Gastin, P. and Oddoux, D. "Fast LTL to Buechi Automata Translation", in *Proc. of the 13th International Conference on Computer Aided Verification (CAV 2001)*. July 2001, Paris, France. Springer, LNCS 2102, pp. 53-65. G. Berry, H. Comon, and A. Finkel, Eds.

[9] Clarke, E.M., Grumberg, O., and Peled, D.A., *Model Checking*: The MIT press, 1999.

[10] Giannakopoulou, D. and Lerda, F., "Efficient translation of LTL formulae into Büchi automata", RIACS, Technical Report, 01.29, June 2001.

[11] Wolper, P. "Constructing Automata from Temporal Logic Formulas: A Tutorial", in *Proc. of the FMPA 2000 summer school*. July 2000, Nijmegen, the Netherlands.